Using inline assembly with IBM XL C/C++ compiler for Linux on z Systems, Part 4: Advanced features

Anh Tuyen Tran

August 12, 2015

The objective of this article is to discuss the advanced features of inline assembly supported by IBM XL compiler for Linux on z Systems. The target audience is software engineers who are interested in going beyond the extent of the optimizations provided by the compiler to fine tune the most performance-sensitive code section of their applications.

View more content in this series

Introduction

IBM XL C/C++ compiler for Linux on z Systems Version 1.1 released in 2015 enables support for incorporating user's assembler instructions directly into C/C++ programs (inline assembly). This provides advanced users with greater flexibility to access instructions at the chip level. With inline assembly, software engineers are able to handcraft assembler codes for the most performance-sensitive parts of C/C++ programs. This can further accelerate the execution of the applications to the full extent of the programmers' ingenuity.

The objective of this article is to introduce the advanced features of inline assembly supported by IBM XL compiler for Linux on z Systems. It discusses in detail assembly labels, basic branching, relative branching, symbolic names for input and output operands, the matching constraints, and the registers on the clobber list. The scope of this article is within the assembler instructions involving general registers. Vectors registers and floating-point registers will be deliberated as a separate issue. The target audience is advanced software engineers interested in going beyond the extent of the optimizations provided by the Linux on Z Systems compiler to fine tune the most performance-sensitive code section of high-performing applications.

More in this series.

Assembly labels

During the compilation process, the compiler creates an internal name in the object file for each variable and function declared in the user's program. The name is also used to refer to the corresponding variable or function in the assembly code. The assembly label feature allows the user to control the internal names in the object file of certain variables and functions. When the assembly code is generated, the name specified by the assembly label is the name for the corresponding variable or function. Thus, the declaration

```
int func( ) asm("my_function")
```
specifies that the name in the object file for function `func` is `my_function`, but not the conventional `_func`.

A possible usage of this feature is to allow users to define names for the linker that do not start with an underscore, even on a system where an underscore is normally prepended to the name of a function or variable. Note that the assembly label specification can only be applied to the declaration of global variables and function prototypes.

The C programs `label_b.c` and `label_a.c` shown in Listing 1 and Listing 2, respectively, are snippets demonstrating the usage of assembly label for a function prototype.

**Listing 1. label_b.c defining the function `func_asm`**

```c
int func_asm() {        //func_asm is defined here
    return 55;
}
```

In file `label_b.c`, the `func_asm()` function is defined.

**Listing 2. label_a.c associating function name with and assembly label**

```c
int func() asm("func_asm");        // func is associated with "func_asm"
int main() {
    return func();                 // func is called
}
```

In file `label_a.c`, function `func` is associated with the name `func_asm` by means of the assembly statement on line 1. On line 3, function `func()` is called, although there is no definition for it. The expectation is that `func` is bound to `func_asm` and the call to function `func()` will become a call to `func_asm()`.

Compiling, linking, and running the executable from `label_a.c` and `label_b.c` should be successful. The execution returns 55 because the symbol `func` is bound to `func_asm`. Figure 1 show the assembly codes generated for program `label_a.c`. It confirms that the name `func_asm` is used in place of `func`: `[ BRASL %r14, func_asm ]`.

**Figure 1. label_a.c calls func_asm instead of func**

![Figure 1](image-url)
Branch to a label

There are two ways to branch to a label: **basic branching** and **relative branching**. In basic branching, the branch instruction branches to a label based on certain conditions. A label must be uniquely defined in a given program. In relative branching, the target label is relative to the location of the branch instruction. If the target label is before the branch instruction, the character b (for backward) is added to the branch address. In the same manner, f (for forward) will be added to the branch address when the target label is after the branch instruction.

### Basic branching

Listing 3 is an example of using basic branching.

**Listing 3. Example of basic branching**

```c
int absoluteValue(int a) {
    asm (" CFI %0, 0\n" " BRC 0xA, DONE\n" " LCR %0, %0\n" " DONE:\n"
        :"r"(a)
    );
    return a;
}
```

Table 1 shows the relationship between condition code and the mask for the instruction CFI (compare immediate) used on line 2 of Listing 3.

**Table 1. Relationship between condition code and mask for instruction CFI**

<table>
<thead>
<tr>
<th>Compare a against 0</th>
<th>Condition code</th>
<th>Mask bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 0</td>
<td>0 = 00₂</td>
<td>1000</td>
</tr>
<tr>
<td>a &lt; 0</td>
<td>1 = 01₂</td>
<td>0100</td>
</tr>
<tr>
<td>a &gt; 0</td>
<td>2 = 10₂</td>
<td>0010</td>
</tr>
</tbody>
</table>

On line 2 of Listing 3, the CFI instruction compares variable a (%0) with zero. If a == 0 or a > 0, the condition code will be set to either 0 or 2 (as per rows 2 and 4 of Table 1). The combined mask bits for condition codes 0 and 2 is 1010₂. In hexadecimal representation, 1010₂ is 0xA.

Accordingly, if a >= 0, the branch instruction on line 3 will branch to the label DONE on line 5. The function returns the value of a without running the LCR instruction on line 4. On the other hand, if a < 0, the branching will not occur. Instruction LCR on line 4 loads the complement of a to itself before returning a. Thus, the function effectively returns the absolute value of a. The basic branching to label DONE in this example is used to skip the execution of LCR on line 4.

### Relative branching

The example in Listing 4 uses relative branching to loop back.
Listing 4. Pseudo code using relative branching

```c
asm ( 
"1:          
 DoSomeWork
 BRCT %0, 1b  
 :"+r"(limit)
 );
```

The `BRCT` (Branch Relative On Count) instruction subtracts 1 from the value of the first operand `limit` (%0) and stores the result back to the operand. When the result is not zero, it branches to the address specified in the second operand, which is `1b`, that is, label `1`-backward. Relative to the branch instruction, label `1` is backward on line 1. In this example, as long as `limit` is not zero, `BRCT` decrements it, and then loops back to label `1`. When `limit` becomes zero, the loop terminates.

Note that for relative branching, the label name must contain numbers only. This requirement is not applicable for basic branching label. Also, the label has to be within the same assembly statement. Jumping to a label in a different assembly statement is not supported.

### Symbolic names

The input and output operands can also be specified by symbolic names. The symbolic names can be referenced within the assembly code. Symbolic names are specified inside square brackets preceding the constraint string. Inside the assembly code, symbolic names can be referenced using `%[name]` instead of a percentage sign followed by the operand number. Symbolic names can be any valid C variable name, even if the names have been defined in the surrounding C code. Symbolic names, however, must be unique within each inline assembly statement.

The snippet in Listing 5 uses symbolic names `[results]`, `[first]`, and `[second]` to represent the 0th, 1st, and 2nd operands respectively. Instead of referring to `%0`, `%1`, and `%2`, the statement will refer to `%[result]`, `%[first]`, and `%[second].

Listing 5. Example of using symbolic names

```c
int main()
{
    int sum = 0, one=1, two = 2;
    asm ("AR  %[result], %[first]\n        "AR  %[result], %[second]\n        :[result] "+r"(sum)
             :[first] "r"(one), [second] "r"(two)
    );
    return sum == 3 ? 0 : 1;
}
```

### Matching constraints

0, 1, ..., 9 are matching constraints used to advise the compiler to allocate the same register for both the input operand and the numbered output operand. As such, the matching constraints can only be used with the input operands. This is essential when one of the operations uses the result of a previous one as its input. Without a matching constraint, the compiler does not know that the same register must be used for both the output and input operands.
The C program example07a.c in Listing 6 is an example where the execution might produce incorrect results due to the absence of a matching constraint.

Listing 6. example07a.c with incorrect result without a matching constraint

```c
#include <stdio.h>
int main () {
    int a = 10, b = 200, c = 3000;
    printf("INITIAL: a = %d, b = %d, c = %d\n", a, b, c);
    asm ("LR %0, %2"
         :="r"(a),="r"(b)
         :r"c", "0"(a));
    printf("RESULT : a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}
```

In the first LR (load registers) instruction on line 5, \texttt{a} is loaded with \texttt{c}. Because \texttt{c} is 3000, \texttt{a} will become 3000. Then comes the second LR instruction on line 6, where \texttt{b} is loaded with \texttt{a}. If the intent of the programmer is to load the updated value of \texttt{a}, which is 3000 after the first LR instruction, then example07a.c will not deliver that result. There is no guarantee that the compiler will use the same register for the same variable \texttt{a} between the two times LR is called. When it does not, the previous value of \texttt{a}, being 10, will be loaded to \texttt{b}. Listing 7 shows that compiling and running example07a.c will yield an incorrect result of \texttt{b} being 10 instead of 3000 in most cases.

Listing 7. Compiling and running example07a.c

```
xlc -o example07a example07a.c;
./example07a
INITIAL: a = 10, b = 200, c = 3000
RESULT : a = 3000, b = 10, c = 3000      <- b is loaded with a, but b is 10 while a is 3000
```

Because the intent of the user is to load the updated value of \texttt{a} to \texttt{b}, matching constraint must be used to indicate to the compiler that the output of LR instruction on line 5 is used as the input of LR instruction on line 6. When the matching constraint is used, the compiler will select the same register for variable \texttt{a} during the execution of both LR instructions. Listing 8 shows the C program example07b.c, which is the corrected version making use of a matching constraint.

Listing 8. example07b.c with a matching constraint

```c
#include <stdio.h>
int main () {
    int a = 10, b = 200, c = 3000;
    printf("INITIAL: a = %d, b = %d, c = %d\n", a, b, c);
    asm ("LR %0, %2"
         :="r"(a),="r"(b)
         :r"c", "0"(a));
    printf("RESULT : a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}
```

The program example07b.c uses the matching constraint \texttt{"0"(a)} on line 8 to inform the compiler that the input operand \texttt{a} \texttt{(%3)} must use that same register as the 0th output operand \texttt{a}. Because
the first LR instruction on line 5 loads c being 3000 to a, and the second LR instruction on line 6 will use the same register for input operand a, the value 3000 will be loaded to b as expected.

Figure 4 displays the difference between the two assembly files generated for example07a.c (on the left side of the figure) and example07b.c (on the right side of the figure). When there is no matching constraint, (example07a.c), it is evident that the compiler uses two different registers r1 and r5 for output operand a and input operand a respectively. When a matching constraint is used in example07b.c, the same register r1 is used for both LR operations.

Figure 2. Code generation depending on the existence of matching constraint

Table 2 explains example07a.c, where a matching constraint is not used. The update that occurred with r1 is independent from the input value on r5. For that reason, the updated value is not used in the second LR instruction.

Table 2. Codes when matching constraint is not used (example07a.s)

<table>
<thead>
<tr>
<th>Assembly codes</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRASL %r14,printf</td>
<td>calls printf INITIAL ...</td>
</tr>
<tr>
<td>L %r3,168(,%r15)</td>
<td>Loads value of c from r15+168 to register r3: r3 holds 3000</td>
</tr>
<tr>
<td>L %r5,176(,%r15)</td>
<td>Loads value of a from r15+176 to register r5: r5 holds 10</td>
</tr>
<tr>
<td>#GS00000</td>
<td>Starts inlining user’s assembler instructions</td>
</tr>
<tr>
<td>LR %r1, %r3</td>
<td>Loads r3 (value of c, being 3000) to r1 (a)</td>
</tr>
<tr>
<td>LR %r0, %r5</td>
<td>Loads r5 (value of previous a, being 10) to r0 (b)</td>
</tr>
</tbody>
</table>
On the other hand, the assembly codes for `example07b.c`, where the matching constraint is used, reveals that for the same register r1 is used for variable a. The update occurred with r1 after running the first LR instruction becomes the input value for the second LR. For that reason b is correctly loaded with the updated value of a.

### Table 3. Codes generated when matching constraint is used (example07b.s)

<table>
<thead>
<tr>
<th>Assembly codes</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRASL %r14,printf</td>
<td>calls printf INITIAL ...</td>
</tr>
<tr>
<td>L %r3,168(,%r15)</td>
<td>Loads value of c from r15+168 to register r3: r3 holds 3000</td>
</tr>
<tr>
<td>L %r1,176(,%r15)</td>
<td>Loads value of a from r15+176 to register r1: r1 holds 10</td>
</tr>
<tr>
<td>#GS00000</td>
<td>Starts inlining user’s assembler instructions</td>
</tr>
<tr>
<td>LR %r1, %r3</td>
<td>Loads r3 (value of c, being 3000) to r1 (a)</td>
</tr>
<tr>
<td>LR %r0, %r1</td>
<td>Loads r1 (value of updated a, being 3000) to r0 (b)</td>
</tr>
<tr>
<td>#GE00000</td>
<td>Ends inlining user’s assembler instructions</td>
</tr>
</tbody>
</table>

#### Register names on the clobber list

If the assembler instruction uses or updates registers that are not listed in the output and input operand lists, the user must list all impacted registers in the clobber list. Based on the information, the compiler facilitates the operations of the inline assembly statement.

Listing 9 displays an example where general register r7 is explicitly specified as the operand of the assembler instructions.

### Listing 9. example09.c using register not in the input/output operand list

```c
#include <stdio.h>

int main () {
    int a = 15, b = 20;
    printf ("INITIAL: a = %d, b = %d\n", a, b );
    asm ("LR   7, %1
           MSR %0,  7\n           ":+r"(a)
           :r"r7"
    );
    printf ("RESULT : a = %d, b = %d\n", a, b );
    return 0;
}
```

The LR instruction on line 5 specifies register r7 as its output operand. The MSR instruction on line 6 also uses r7 as its input operand. Register r7 is used as an operand of the assembler instructions, but it is not listed on the input and output operand list. For that reason, r7 must be added to the clobber list to inform the compiler that it is used. In general, to ensure the correctness of the program any register affected by the assembler instruction must be listed either in the operand lists or in the clobber list. The compiler relies on the information to adjust register allocation.
Comparing the difference in the codes when altering the register in use exposes how the clobbering of certain registers impacts the performance. In the example09a.c program exhibited in Listing 10, register r1 is used instead of register r7.

**Listing 10. example09a.c clobbering a different register**

```
#include <stdio.h>
int main () {
  int a = 15, b = 20;
  printf ("INITIAL: a = %d, b = %d\n", a, b);
  asm ("LR   1, %1
        MSR %0, 1\n        ":+r"(a)
        :"r"(b)
        :"r1"
      );
  printf ("RESULT : a = %d, b = %d\n", a, b);
  return 0;
}
```

Figure 3 compares the two assembly files generated by the compiler. The file on the left side uses r7 and the file on the right side uses r1.

**Figure 3. Comparing the codes when clobbering different registers**

The right side of Figure 3 shows that when register r1 is clobbered, the compiler selects register r3 for variable \texttt{b} \texttt{[L \%r3,168(,%r15)]}. More importantly, the compiler does not save the contents of the clobbered register when r1 is selected. When r7 is selected, the content of register r7 is saved to the location R15+56 \texttt{[STG \%r7,56(,%r15)]}. This means, clobbering register r1 instead of r7 reduces one STORE instruction. Figure 6 proves that selecting proper register to clobber might improve the performance.

If explicitly specifying a register is not preferred, users can modify the code so that the compiler will be responsible for selecting the correct register. In this particular example, the user can add
a temporary register operand and use a matching constraint to facilitate the fact that the operand is used as both input and output operands. The specific code for `example09b.c` is displayed in Listing 11.

### Listing 11. Code modification to let the compiler select register

```c
#include <stdio.h>

int main () {
    int a = 15, b = 20, tmp = 1;
    printf("INITIAL: a = %d, b = %d\n", a, b);
    asm ("LR %1, %2
        MSR %0, %3"
         :"+r"(a), "=r"(tmp)
         :":r"(b), "1"(tmp)
    );
    printf("RESULT : a = %d, b = %d\n", a, b);
    return 0;
}
```

### Conclusion

Inline assembly provides an avenue for users to incorporate assembler instructions directly into C/C++ programs. This feature allows advanced users to further improve the performance of the applications by handcrafting the assembler instructions for particular sections of the codes. IBM XL compilers perform highly sophisticated tasks to optimize the codes generated at each level of optimization. For that reason, accelerating performance with inline ASM requires the intrinsic knowledge of the user about the execution of the target codes. Careful analysis about the effects on the performance of the embedded assembler instructions, together with thorough planning and testing are the prerequisites for achieving performance gain.

### Acknowledgements

I would like to thank Ms. Visda Vokhshoori and Ms. Nha-Vy Tran for their advice during the composition of this article.

### Resources

- Visit the IBM XL C/C++ for Linux on z Systems product pages for more information.
- Get connected. Join the Rational C/C++ Cafe community.

### References

- z/Architecture Principles of Operation, IBM Publication No. SA22-7832-10
- IBM z/Architecture Reference Summary, IBM Publication No. SA22-7871-08
- Inline assembly statements for XL C/C++ for Linux on z Systems, V1.1, Retrieved June 1, 2015